Spring 5-1-1997

A Performance Evaluation of the Hemingway DSM System on a Network of SMPs ; CU-CS-837-97

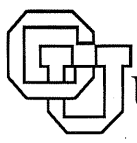
Anshu Aggarwal
University of Colorado Boulder

Dirk C. Grunwald

A Performance Evaluation of the Hemingway DSM System on a Network of SMPs

Anshu Aggarwal
Dirk Grunwald

CU-CS-837-97



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Performance Evaluation of the Hemingway DSM System on a Network of SMPs

Anshu Aggarwal and Dirk Grunwald
Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
(Email: {anshu,grunwald}@cs.colorado.edu)

Abstract

Numerous designs for software distributed shared memory systems have been proposed. Most designs use uniprocessor workstations as the building blocks. In recent years there has been an increase in commodity multiprocessor workstations, with hardware-maintained internal memory coherence mechanisms. In this paper we investigate the performance of a software distributed shared memory system, Hemingway, which is built out of such multiprocessor workstations, utilizing off-the-shelf communication networks.

The effectiveness of this system can be evaluated by studying performance as a function of both the total number of processors in the system and the degree of clustering (size of multiprocessor workstations). We evaluated the performance of Hemingway with systems of upto 8 processors, with different levels of clustering. We also compared the performance of our protocol with a similar, established protocol, the Munin protocol.

Our results describe a system that scales well both with the number of processors and with clustering. Moreover, our studies indicate that the Hemingway protocol requires lower intra-workstation and inter-workstation network bandwidths than other protocols. Overall we have found that clustering is very effective in increasing performance in software DSM systems built with multiwriter, write-through memory consistency policies.

1 Introduction

The traditional approach to building large scale shared memory machines has been to design special-purpose, monolithic systems. A reduction in the demand for such specialized machines, along with an increase in the computational power of commodity workstations, has led designers to consider alternatives to monolithic systems, such as software distributed shared memory systems (DSMs), which are built out of networked workstations. Numerous designs for these have been proposed to date, many of which have actually been realized. Most of the designs, however, concentrate on the use of uniprocessor workstations as the fundamental building block. The recent increase in the availability of commodity multiprocessor workstations, however, raises an interesting question: can a software distributed shared

memory system built out of networked multiprocessor workstations take advantage of the internal processor clustering to improve overall system performance? We address this issue in this paper.

Memory coherence operations and memory transfers are the main sources of the differences in latency between different software DSM designs. Processor clustering is of benefit if it leads to a reduction in latency. The latency is dependent upon the amount of necessary memory coherence, the coherence policy implemented and the available network bandwidths.

Multiprocessor workstations have hardware support to provide a small-scale shared memory environment. An n-processor software DSM system built out of n uniprocessor workstations has to keep the memory coherent between all n processors. In contrast, an n-processor software DSM system built out of m $\frac{n}{m}$-processor SMP workstations has to keep the memory coherent only between m SMPs. The internal SMP hardware coherence mechanism provides coherence between the $\frac{n}{m}$ processors in each SMP. A design with SMPs, therefore, is likely to result in a decrease in the amount of necessary memory coherence.

The choice of a memory consistency policy has a very major effect on memory coherence latencies. Although many policies have been described, release consistency offers lowest overall latency [14]. Release consistency can be implemented in many different ways - with a single-writer implementation as in DASH [17], SoftFLASH [10] and Shasta [20], or with a multi-writer implementation as in Munin [3], Treadmarks [7], Shrimp [6], Cashmere [21] and Hemingway [2]. Multi-writer implementations can be further implemented with a write-back scheme as was done with Munin and Treadmarks, or with a write-through scheme as in Shrimp, Cashmere and Hemingway. Single-writer systems can suffer from false-sharing [9]. Multi-writer alleviates false sharing by allowing several simultaneous write-sharers, reserving coherence for specific points in time. Write-back and write-through implementations of multi-writer policies differ in when the memory becomes coherent and in the amount of computation required at memory coherence points. It has been argued that while write-back systems are computationally and latency intensive at memory coherence points, write-through systems may use more overall network bandwidth and yield poor performance. We have found that the high memory coherence latency of write-back systems is a larger detriment to performance than the increased network bandwidth usage in write-through systems.

Communication network bandwidth comes in two flavors: bandwidth within the fundamental building block of a software DSM, such as the system bus bandwidth, and the bandwidth between the fundamental blocks, the interconnect bandwidth. If the effective bandwidth available to each processor in an SMP is to remain constant with clustering, both the internal and external bandwidths per SMP have to scale linearly with the number of processors per SMP. Existing SMP workstations, however, do not necessarily provide this. Software DSMs built out of multiprocessor workstations, then, can suffer from inadequate available network bandwidths. We have studied the amount of bandwidth necessary for comparable performance in SMP systems and found that it does not need to scale with the number of processors.

In order to concretize the study described above, we compared the performance of the Hemingway protocol with that of the Munin protocol on different sized systems, with different levels of clustering. The comparison between Hemingway and Munin is very important because the two protocols affect memory coherence latency and network bandwidth usage differently. In the sections that follow we describe the Hemingway protocol, our experimental framework and our results. After an analysis of our results we conclude that small-scale SMPs are good building blocks for software DSM systems because they allow processor clustering to replace expensive inter-SMP communication by cheaper intra-SMP communication, leading to an overall increase in performance. The network bandwidth requirements do not increase as fast as the number of processors per SMP, allowing existing commodity SMP workstations to be used for building distributed shared memory systems. These results are as a consequence of the multi-writer, write-through protocol of the Hemingway system.

2 Hemingway

The Hemingway system uses a multi-writer, write-through implementation of Eager Release Consistency to provide a coarse-grain distributed shared memory environment. The write-through implementation requires the use of a memory-mapped communication network for low-overhead communication.

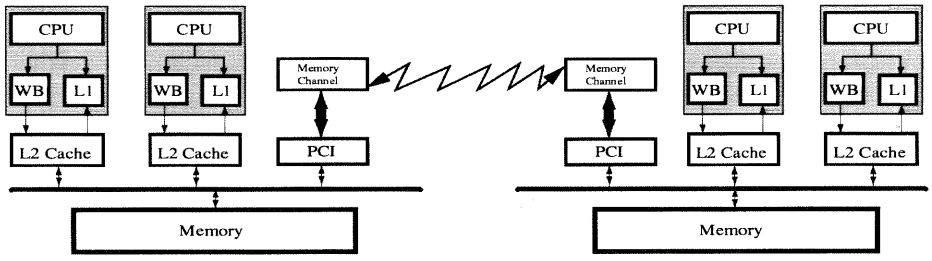


Figure 1: System components in experimental architecture.

The basic idea behind Hemingway is to associate a "home" with every unit of shared memory. Writers to the unit get the most up-to-date copy from the home and update the home with modifications, as they occur. Release consistency requires that modifications become globally visible by the time of lock acquire operations. This places an ordering constraint on the communication, requiring that all modifications made within critical sections reach the corresponding homes before the next entry into a critical section protected by the same lock. This is a standard requirement of all multi-writer home-based systems. Processors entering critical sections invalidate local copies of shared units, if they are not the homes of the units. There is an optimization to this protocol, where the local copies do not need to be invalidated when the processor acquiring a lock was the last releaser of the lock. A further optimization can be made with information about which units were accessed within which critical sections, to further reduce the number of invalidations. We have started an implementation of this second optimization, but it is not far enough along yet to allow us to report results from it in this version of the paper.

Hemingway does not impose a requirement on the mechanism to control access to shared memory. The sharing granularity, therefore, depends on the mechanism chosen. In implementations where operating system support for virtual memory is used to control accesss, the sharing granularity is limited to the supported page size. This is not a problem since multi-writer protocols alleviate the false-sharing problem endemic to single-writer systems, allowing coarse-grain sharing. Potential for symbiotic sharing between processors on an SMP, in fact, suggests coarse-grain sharing.

3 Experimental Framework

We constructed a cycle-level simulator, based on Aint [18], to conduct our study. In addition, we also began an implementation of the Hemingway protocol using the existing framework for the Shasta system [20]. The Shasta system uses inline-checks in application programs to supports fine-grained sharing. At present we are restricting our study to sharing at an 8K byte granularity, for compatability with our simulation studies. The Hemingway protocol may benefit from finer-grained sharing and we intend to conduct that study once the implementation is complete. We describe the simulation and experimental frameworks next.

3.1 Simulation

The basic system that we simulate consists of Digital Alpha 21064 SMPs connected together by Memory Channel [11], as in Figure 1. The Memory Channel is a low-latency memory-mapped interconnection network, similar to VAXClusters [16], Shrimp [6] and Hamlyn [23]. Memory channel allows processors to map pages into their virtual address space so that stores to any location on such pages are "reflected" into the physical address space of remote processors. This mapping capability can be used very effectively to support write-through. All shared writes can automatically update the home, without operating system intervention, once the initial mapping is established. Memory Channel unfortunately does not prvide similar support for remote reads. Remote memory pages are demand-fetched, requiring signalling and interruption of remote processors in order to do the transfer. All of the shared memory pages are

interleaved across the SMPs, for example on a two-SMP system sharing four pages, the first SMP would "own" pages 0 and 2, while the second SMP would own pages 1 and 3. More intelligent or adaptive policies for page placement and ownership transfer are possible, but are not considered in this paper.

We use the system parameters defined for the DECchip 21064 [8] for our study. In order to contain the memory requirements of the simulator, we decreased the size of the L2 cache from 4Mbytes to 2Mbytes. This is large enough to contain the important working sets of the applications we simulate [24]. All other system parameters were left unchanged. The table in Figure 3.1 summarizes the main parameters in our study. These parameters define the

Processor	275MHz Clock ; Single-Issue
Page Size	8K Bytes
Write Buffer	4 32 Byte Blocks 1 Cycle Access Time 2-Entry/256 Cycle Flush Threshold
L1 Cache	8K Bytes, 32 Byte Blocks - Direct Mapped 1 Cycle Access Time
L2 Cache	2M Bytes, 32 Byte Blocks - Direct Mapped 2 Cycle Access Time 4 Cycle Cycle Time
BIU	128 Bits Wide
TLB	32 Entries - Fully Associative 1 Cycle Access Time 2000 Cycle Invalidation Time
Page Table	35 Cycle Access Time 70 Cycle Miss Penalty
DRAM	10 Cycle Access Time 20 Cycle Cycle Time
System Bus	Bandwidth: 640MB/s - 4GB/s
Memory Channel	Bandwidth: 100MB/s - 800MB/s Receive Buffer Size: 8K - 64K Bytes Transmit Buffer Size: 8K - 64K Bytes PCI Access Latency: 17 cycles

Figure 2: System parameters.

hardware and operating system characteristics the simulated systems are based on. The actual simulated times to complete load and store instructions is a factor of network and resource contention in addition to the fixed delays listed in Figure 3.1.

We modeled the complete memory system in great detail starting with the on-chip L1 cache and write-buffer, the TLB, the second level cache, the DRAM, the page table and the Memory Channel buffers. The simulations model contention at each level of the memory and network hierarchy: the on-chip bus interface unit (BIU), the L2 cache interfaces, the system memory bus, the bulk memory, the page table, the Memory Channel interfaces and the Memory Channel networks. All coherence, signalling and communication traffic contends for shared resources. Inter-processor (intra-SMP) communication occurs over the shared system bus, and inter-SMP communication ties up the system buses and the appropriate Memory Channels locally and remotely. The simulator was verified by running in "Protocol Verification Mode" [22], where all of the load and store values were actually stored in the simulator and supplied by it to the executing application. Since instruction interlocks and latencies were not modeled, the inter-arrival times for memory requests are closer than they would be in practice, placing a greater strain on the memory subsystem. We assume we can fetch remote pages without interrupting the processor; this benefits systems using many small transfers more than the page-based system we propose.

3.2 Implementation

Our implementation base is the existing Shasta system [20]. Shasta is a software DSM system, with support for fine-grained sharing. Shasta instruments application executables, inserting inline-checks before all load and store instructions to control access. Non-local loads and stores invoke protocol functions for transfer of data. Shasta was developed with a single-writer implementation of eager release consistency. The protocol initially supported shared memory over a network of uniprocessors. It was modified to support an implementation over a network of SMPs [19]. We used this latter implementation to start building a multiwriter system. The current system supports multi-writer with write-back, requiring the computation of "diffs" to determine local changes to shared pages. The diffs update the homes of the shared pages at lock **release** time. Shared pages are invalidated at lock **acquire** time. The implementation has so far only been verified on simple test cases. Once the write-back implementation is complete, we will start the implementation of the write-through protocol.

The simulation and the implementation differ in three fundamental aspects:

- the implementation uses inline-checks to control access to shared memory, whereas the simulator assumes the use of the operating system virtual memory support,
- the simulator assumes that there exists a framework for fetching remote pages without interrupting remote processors, whereas the implementation requires support from remote processors to do page transfers and
- the simulator does not model instruction latencies.

It is important to understand that these differences do not violate any of our results, because both the simulation and implementation frameworks are stand-alone and used for comparative studies. We use the simulator to do a parametric exploration of the design space of system size, clustering and communication bandwidths. The implementation is used to validate our design and to conduct further studies on the effects of sharing granularity on performance.

3.3 Measurements for Study

The goal of our study is to determine the effectiveness of using SMPs as building blocks for a software DSM system. We define effectiveness to require that:

Processor Scaling: The performance of the system increase with the total number of processors in the system,

Clustering Scaling: The performance of the system increase with clustering - the size of the constituent SMPs, keeping the total number of processors in the system, constant and

Bandwidth Scaling: The network bandwidth requirements do not increase faster than linear.

In the sections that follow we present results from our experiments in the context of the metrics defined above. We compare the performance of the Hemingway protocol with a variation of the Munin protocol. The standard Munin protocol reserves memory coherence operations for specific synchronization points, at which time "diffs" of local modifications are computed and sent to sharers. We study a modification of this protocol where only homes of pages are updated at the synchronization points. Sharers needing up-to-date copies of pages can then get them from the home. Both the Hemingway and Munin protocols were implemented with the optimization where shared pages are not invalidated if the lock-acquiring processor was the last lock-releaser. This optimization is extended in SMPs so that pages do not need to be invalidated when a lock-acquring processor is on the same SMP as the lock-releasing processor.

4 Results

The medium of our study is twofold: implementation-based and simulation-based. The simulation studies are not limited by the fixed hardware-base of the implementations and therefore allow the study of the effects of hardware variations on performance. The positive results from the simulations have motivated the implementation, which was begun recently and is not yet very far along. We have so far only verified its correctness on simple test cases.

We defined three metrics to study the effectiveness of SMPs as building blocks for software DSMs in §3. They are, briefly, scaling with total number of processors (Processor Scaling), scaling with clustering (Clustering Scaling) and network bandwidth requirements (Bandwidth Scaling). We conducted our studies on six applications, four from the SPLASH-2 benchmark suite [24] (Water, LU Decomposition, Barnes-Hut and Ocean) , a multithreaded version of matrix multiply and a multithreaded version of Red-Black Single Order Relaxation. We kept the problem sizes constant as the hardware configuration was changed. The application characteristics are:

1. Matrix Multiply : 512 × 512 matrices, allocated linearly,
2. Water: 1000 molecules run for 10 timesteps,
3. Red Black SOR: 512 element grid, with 20 timesteps,
4. LU Decomposition: 512 × 512 matrix.
5. Barnes-Hut: 16K particles,
6. Ocean: 512 bodies partitioned contiguously and

4.1 Simulation Results

We simulated the execution of the applications listed above on systems with different total number of processors, with different levels of clustering. The effective scalabilities of the Hemingway and Munin protocols can be seen from graphs of the total execution times - Figures 4.1, 4.1 and 4.1. We address the three quantification metrics listed above, separately, in the sections that follow.

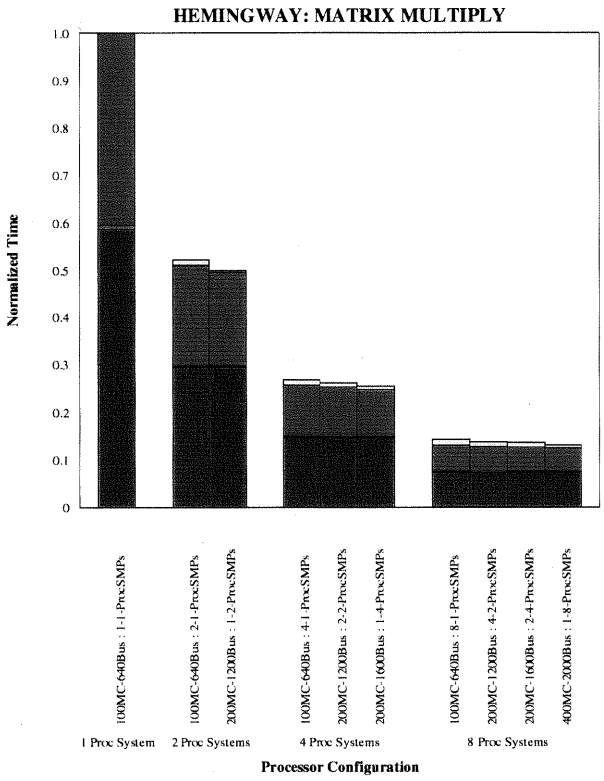
The figures graph the execution time for different processor configurations, normalized against the time for a uniprocessor run. The X-axis represents the processor configurations, clustered with different total numbers of processors. For example, "8 1-processor SMPs", "2 4-processor SMPs", "4 2-processor SMPs" and "1 8-processor SMP" all represent 8 processor systems, with different levels of clustering (increasing left to right). Each figure has four graphs, the top two are for two different applications run with the Hemingway protocol, the bottom two with the same applications run with the Munin protocol. A legend, representative of all graphs, appears in the top right graph in each figure.

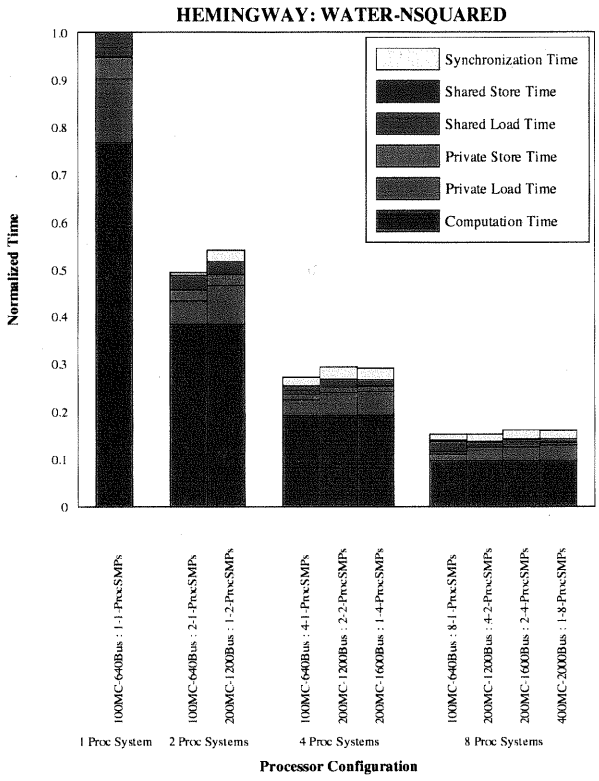
The total execution time is broken up into the fraction of the time spent in the actual computation (*Computation Time*), the time spent completing the non-shared memory reference operations (*Private Load Time* and *Private Store Time*), the time spent completing the shared memory operations (*Shared Load Time* and *Shared Store Time*) and the time spent synchronizing (*Synch Time*), which includes the TLB and PTE invalidation times. Each of these times represent the total time that the processor spends stalled between when an instruction is issued to when it completes.

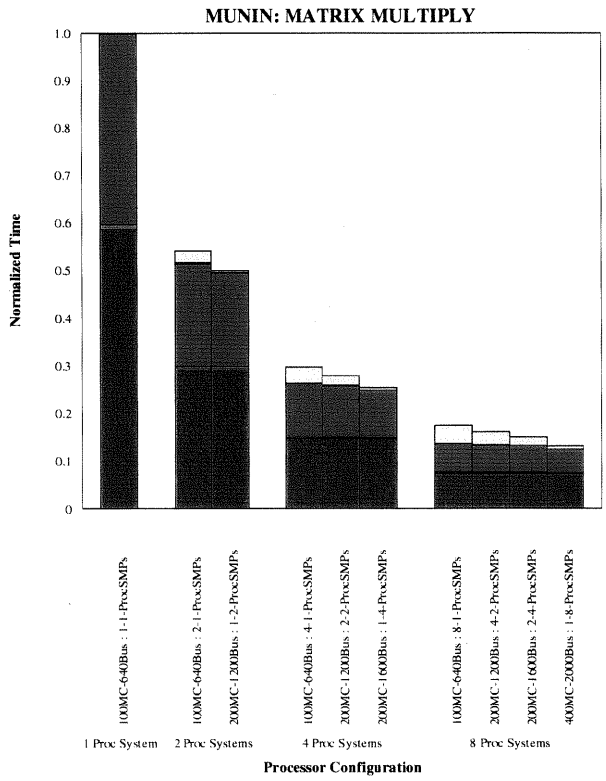
Note: Some of the graphs are incomplete. This is because the simulation runs for those configurations have not yet completed. The very high level of detail in the simulations leads to an average execution time of between 15 - 20 days for each run.

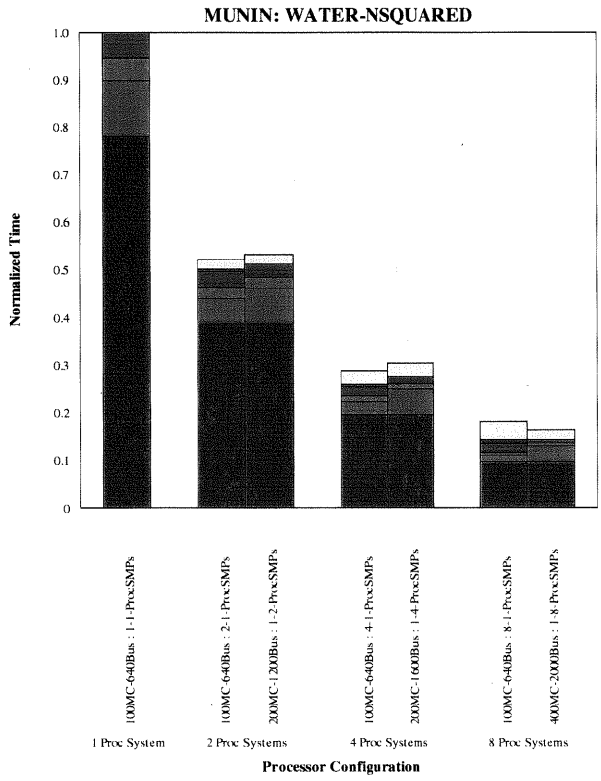
4.1.1 Processor Scaling

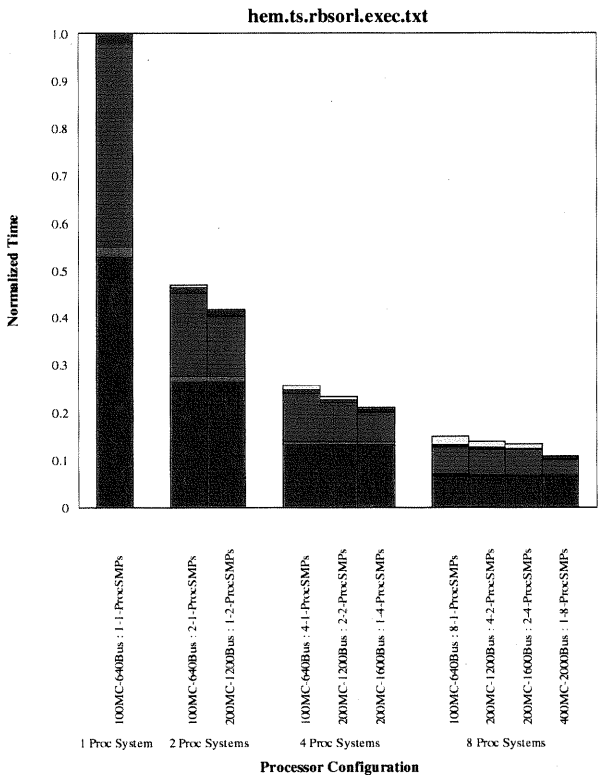
The leftmost bar in each cluster of each graph depicts the execution on a system of networked 1-processor SMPs (uniprocessors). Comparing just these bars between clusters we note that Hemingway scales almost linearly for Matrix

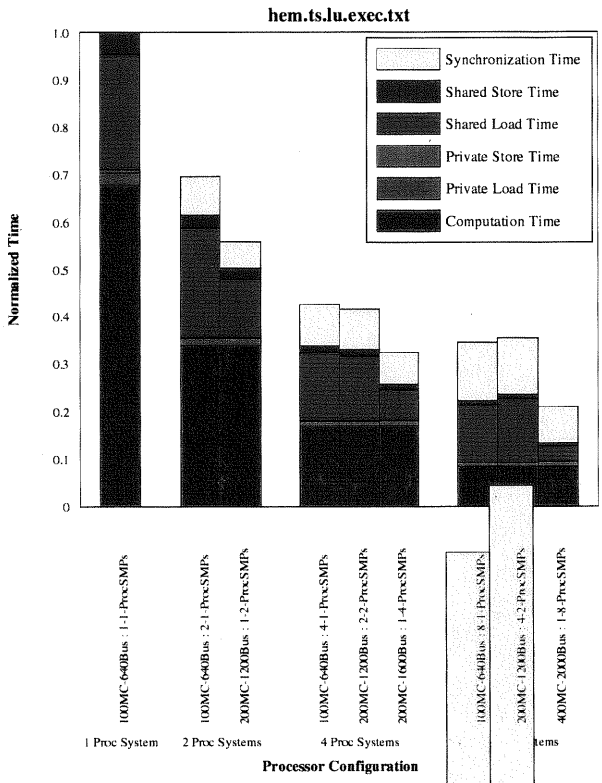

HEMINGWAY: MATRIX MULTIPLY
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1200Bus : 2-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
200MC-1600Bus : 2-4-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration

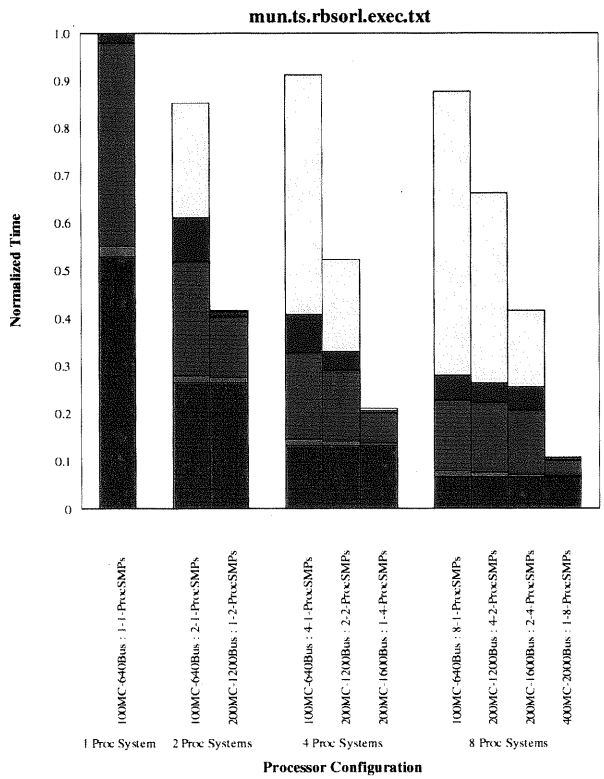

HEMINGWAY: WATER-NSQUARED
Synchronization Time
Shared Store Time
Shared Load Time
Private Store Time
Private Load Time
Computation Time
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1200Bus : 2-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
200MC-1600Bus : 2-4-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration

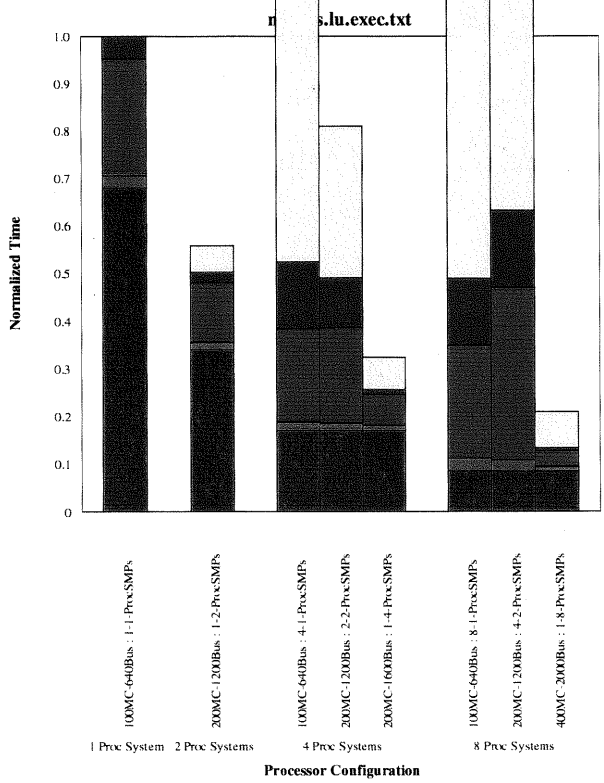

MUNIN: MATRIX MULTIPLY
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1200Bus : 2-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
200MC-1600Bus : 2-4-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration

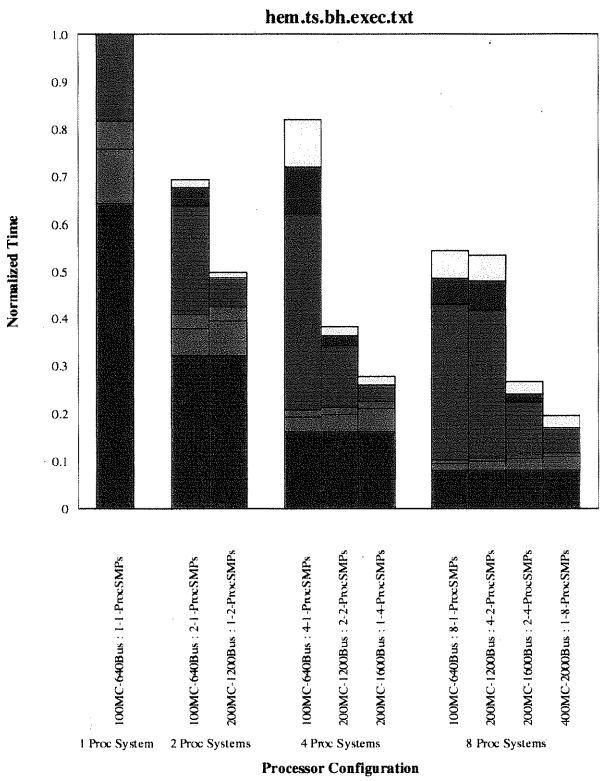

MUNIN: WATER-NSQUARED
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration

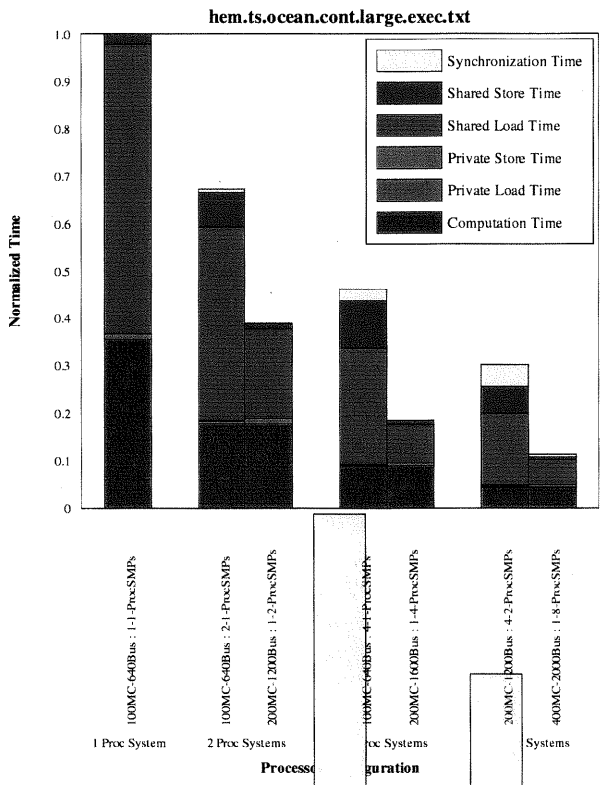

hem.ts.rbsorl.exec.txt
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1200Bus : 2-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
200MC-1600Bus : 2-4-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration

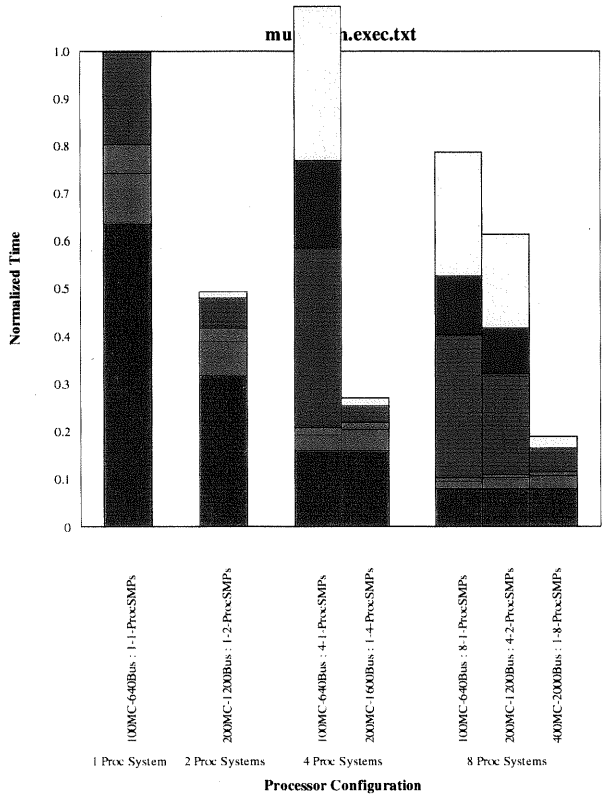

hem.ts.lu.exec.txt
Synchronization Time
Shared Store Time
Shared Load Time
Private Store Time
Private Load Time
Computation Time
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1200Bus : 2-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
Processor Configuration

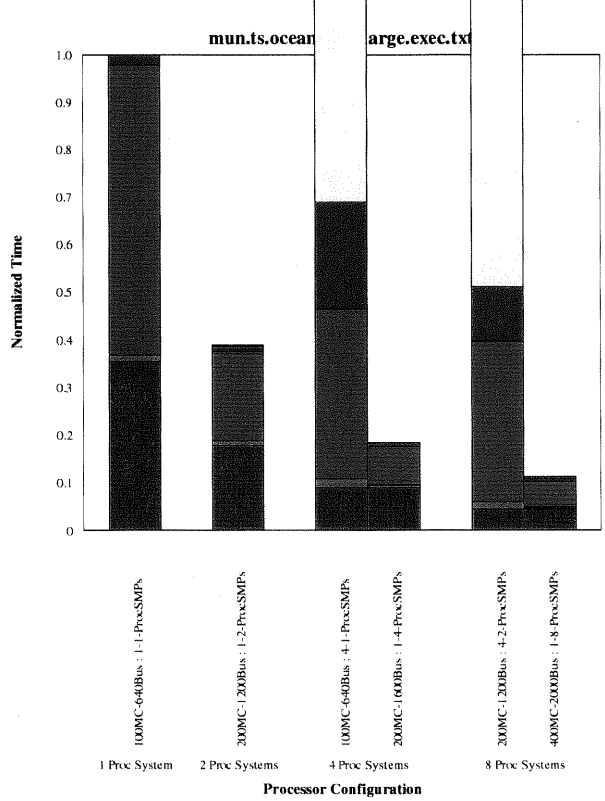

mun.ts.rbsorl.exec.txt
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1200Bus : 2-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
200MC-1600Bus : 2-4-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration


s.lu.exec.txt
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1200Bus : 2-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration


hem.ts.bh.exec.txt
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1200Bus : 2-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
200MC-1600Bus : 2-4-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration


hem.ts.ocean.cont.large.exec.txt
Synchronization Time
Shared Store Time
Shared Load Time
Private Store Time
Private Load Time
Computation Time
Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
100MC-640Bus : 2-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems


Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
100MC-640Bus : 8-1-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration


Normalized Time
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
100MC-640Bus : 1-1-ProcSMPs
200MC-1200Bus : 1-2-ProcSMPs
100MC-640Bus : 4-1-ProcSMPs
200MC-1600Bus : 1-4-ProcSMPs
200MC-1200Bus : 4-2-ProcSMPs
400MC-2000Bus : 1-8-ProcSMPs
1 Proc System
2 Proc Systems
4 Proc Systems
8 Proc Systems
Processor Configuration

Multiply, Water and Red Black SOR. The speedups are lower for LU Decomposition, Barnes-Hut and Ocean. The peformance of Barnes-Hut actually decreases in going from 2 uniprocessors to 4 uniprocessors. This is because of the very large portion of the total time spent in satisfying loads to shared memory. The most latency intensive portion of the Barnes-Hut algorithm is the parallel tree build phase, which has extremely fine-grain locking to build the distributed shared octree. This leads to a very large number of page invalidations at each lock acquire operation - reflected in the relatively larger portion of total execution time spent synchronizing. The pages which are invalidated are then fetched within subsequent critical sections, leading to a large total number of page transfers. We found that the number of pages transferred increased from about 100K to about 500K in going from a 2 uniprocessor system to a 4 uniprocessor system. Increasing the total number of processors in the system without increasing clustering, results in more page invalidations and more page transfers. This affects synchronization time and shared load time. The performance of Barnes-Hut, therefore, does not scale solely with an increase in the total number of processors.

The Munin protocol also shows an almost linear increase in performance for Matrix Multiply and Water. The performance of Munin degrades, however, for the other applications. In all cases this is because of the prohibitively high synchronization costs. Munin reserve the update of homes of shared pages for memory synchronization points, when diffs are sent to homes. This is a latency intensive operation. A comparison of the performance of the Hemingway and Munin protocols on Barnes-Hut with the 4 uniprocessor system (4 1-processor SMPs) can be used for a very clear understanding of the poor performance of the write-back protocol. The total average number of synchronization intervals in Barnes-Hut is about 140,000. The average number of words that are written-through in each interval, corresponding to the average number of modifications made within each interval, was measured to be about 30 (with the Hemingway protocol). The result of the diff operations in Munin, therefore, will yield very few modifications that have to be communicated. The large synchronization time in Barnes-Hut, therefore, provides a good illustration of the high computational cost of diffs.

A comparison of the performance of the two protocols for LU Decomposition also sheds light on the effect of delaying updates for synchronization operations. Hemingway's performance on LU improves with the total number of processors, but the improvement decreases with larger numbers of processors. Looking at the 2, 4 and 8 uniprocessor cases, we see that the total time spent synchronizing increases. This is because the total number of synchronization intervals increases from about 550 to 1300 to 3000. The total number of page transfers, however is very small, but roughly doubles from about 8K to 15K to about 25K in going from 2 uniprocessors to 4 uniprocessors to 8 uniprocessors. The cost of the synchronization operations in Munin, therefore, should not increase much faster than linearly. From the graph in Figure 4.1, however, we see that this is not the case. The answer lies in the number of modifications that have to be made and which have to be communicated back to the homes. We measured the total number of words written through to be about 20M for the 2 processor case, 30M for the 4 processor case and about 45M for the 8 processor case. In Munin these modifications have to be communicated across at synchronization time, leading to intense network congestion and delays. The high synchronization time in LU Decomposition, therefore, provides a good illustration of the high communicational cost of diffs.

The overall conclusion, therefore, is that the performance of the applications does improve with the number of processors in the system, for the Hemingway protocol. This is not true for the Munin protocol because of the increasingly higher cost of synchronization which is due to 1) the computation of a larger number of diffs and 2) the communication of these diffs, which results in heavy network congestion and delays. Hemingway, by spreading out the communication of the modifications, avoids network congestion, reduces coherence costs and scales with the number of processors.

4.1.2 Clustering Scaling

There is a general trend in the performance of both protocols on all applications - the performance increases in each cluster of bars, going from left to right in Figures 4.1, 4.1 and 4.1. The leftmost bar in each cluster represents systems with networked uniprocessors; the rightmost bar, single n-processor SMPs. The performance of all applications is highest on the single n-processor SMPs, with the exception of Water. This, however, is because of bus bandwidth limitations. A single SMP represents total clustering - all processors are in the same shared memory multiprocessor.

The interesting question is whether there is a general improvement in performance with an increase in clustering - going from systems of networked uniprocessors to systems with networked 2-processor SMPs, to systems with networked 4-processor SMPs to systems with networked 8-processor SMPs.

Clustering does not lead to a large performance increase for Hemingway in Matrix Multiply, Water and Red Black SOR. This is because the uniprocessor system performances are very close to optimal - with near linear speedups. There is a marked increase in the performance in Barnes-Hut and Ocean. The increase comes as a consequence of a large decrease in time spent in the shared load and store operations (red and pink sections in bar graphs). This in turn can be attributed to a very large decrease in the total number of pages transferred. The 4 uniprocessor system executing the Hemingway protocol on Barnes-Hut had about 500K page transfers, the 2 2-processor system had about 100K - a decrease by a factor of five. The change is even greater for the 8 processor systems - going from about 650K for the 8 uniprocessor system to about 100K for the 2 4-processor system. The performance of LU Decomposition (Figure 4.1), however, does not change much. This is because the number of page transfers does not change much with clustering, indicating poor increased symbiotic sharing.

The reduction in the number of page transfers is strictly as a consequence of clustering and is not dependant upon the memory coherence protocol used. The total execution time, however, depends on the time spent satisfying memory requests and the time spent synchronizing. These are dependant upon the memory coherence mechanism and available effective network bandwidths. The trends in the performance of the Munin protocol are similar to those of the Hemingway protocol. The overall performance of Munin, however, is much lower. This is because of the higher synchronization costs and the higher memory operation costs. Since the number of pages transfers is similar for both Hemingway and Munin, the higher memory operation costs are due to a difference in the available network bandwidths as a consequence of the difference in the memory coherence mechanisms. This again is a very clear indication of the superiority of a write-through protocol over a write-back protocol.

The overall conclusion, therefore, is that the performance of the applications increases with clustering for both protocols. The increase is directly as a result of the reduction in the total number of page transfers, which in turn is due to effective sharing between processors on the same SMPs. The overall performance of the applications, however, is better with a write-through protocol than with a write-back protocol because of better network bandwidth usage.

4.1.3 Bandwidth Scaling

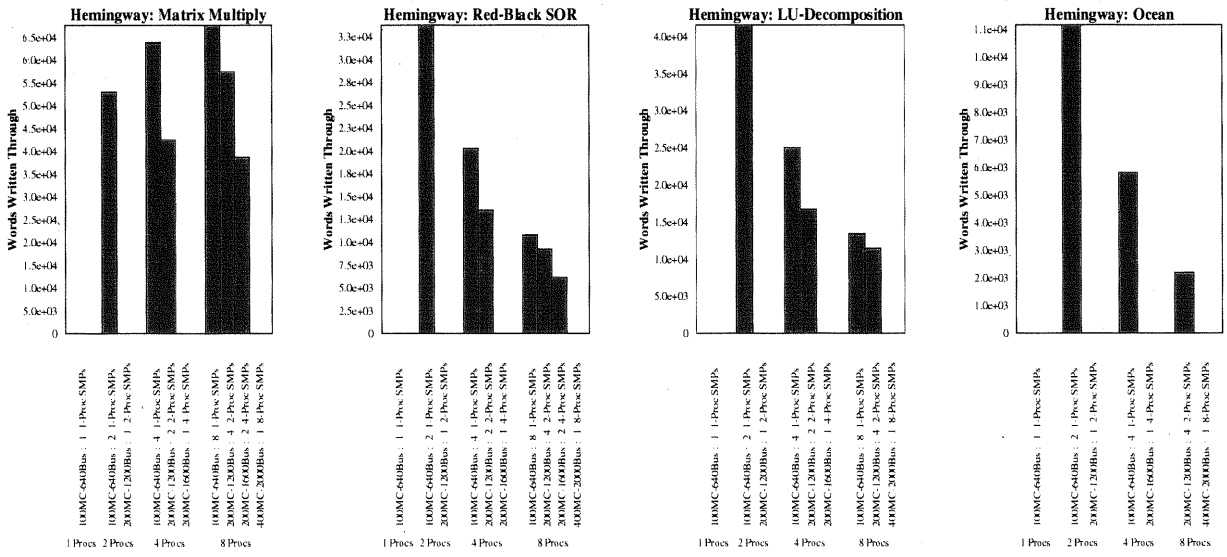


Figure 3: Write-through traffic in some applications.

The main argument against a write-through protocol is that it may result in excessive network bandwidth usage. This is because modifications are written-through as they are made, which may result in multiple modifications to the same locations to be written through, when only the final one matters. The overall effectiveness of a write-through protocol over a write-back protocol is clearly evident from the graphs in Figures 4.1, 4.1 and 4.1, and from the discussion in the previous two sections. The write-through protocol, Hemingway, outperforms the write-back protocol, Munin, for all applications, for all processor configurations.

In order to quantify the write-through traffice, we measured the average number of words written-through per synchronization interval for all applications. The numbers varied with the processor configurations, but Water and Barnes-Hut had at most 70 words/synch interval. This is too small to have an effect on the network. The number of words written through in the other applications are graphed in Figure 3. The write-through traffic affects both the intra-SMP and inter-SMP network bandwidths. The traffic is large for some processor configurations in the applications graphed in Figure 3, but does not have much of an effect on the total memory operation times, as evident from Figures 4.1, 4.1 and 4.1.

Given then, that the Hemingway protocol makes better use of network bandwidths, the interesting issue that has to be addressed is whether the network bandwidth requirements scale linearly with the size of the clusters. In other words, if uniprocessors come supplied with a 640MB/s bus and are connected together by 100MB/s Memory Channel interconnects, do 2-processor SMPs need to have a 1280MB/s bus and 200MB/s Memory Channels and do 4-processor SMPs need to have a 2560MB/s bus and 400MB/s Memory Channel interconnects? The results by Erlichson [9] seem to suggest so, but in the context of a single-writer memory consistency policy. Our results, however, indicate that while the network bandwidths do need to increase, they do not need to increase linearly with SMP size. We ran the above set of applications on 8-processor systems with different levels of clustering with different intra-SMP and inter-SMP network bandwidths. The results for Barnes-Hut, plotted in Figure 4 were typical. The bars are clustered, representing different-sized clusters. Within each cluster the different bars represent runs with different network bandwidths, for example, the 2nd bar from the left plots the execution time for a system of 8 uniprocessors, each with a 640MB/s bus, connected together by a 100MB/s Memory Channel network.

Ignoring the leftmost bar, which plots the uniprocessor run used for normalization, the leftmost cluster represents systems with 1-processor SMPs (uniprocessors). We can ignore this cluster because our starting base is uniprocessor workstations with a 640MB/s bus and 100MB/s Memory Channel. The second cluster represents 2-processor SMPs. Simply increasing the bus bandwidth from 640MB/s to 1.2GB/s (1st and 2nd bars from left in cluster) increases performance significantly. A further increase in bandwidth to 1.6GB/s (3rd bar) does not improve performance much more. An increase in the Memory Channel bandwidth, alone from 100MB/s to 200MB/s (1st and 5th bars in cluster) increases performance significantly. A further increase to 400MB/s (9th bar) produces no change. 2-processor SMPs, therefore, with a 1.2GB/s bus and 200MB/s Memory Channel are effective. A similar analysis of the third cluster from the left indicates that 4-processor SMPs with a 1.6GB/s bus and 200MB/s Memory Channel and 8-processor SMPs with a 2GB/s bus are effective in increasing performance significantly.

As can be seen the effect of the inter-SMP bandwidth decreases with SMP size, because of the decrease in inter-SMP communication traffic. The overall conclusion is that as clustering increases and sharing increases within clusters, the inter-cluster bandwidth becomes less important, but the intra-cluster bandwidth becomes more important. Our results, however, suggest that a small increase in the intra-cluster network bandwidth, relative to cluster size (doubling bus bandwidth for four-fold increase in size or doubling bandwidth for 8-fold increase in size) can improve performance by as much as 30% to 50%.

5 Related Work

Distributed shared memory (DSM) systems may be categorized on the basis of the memory consistency protocol they are implemented with. They may be further divided into single-writer, multi-writer with write-back or multi-writer with write-through implementations of the protocol. Single-writer systems can suffer from excessive false-sharing if the granularity of the sharing is large or from excessive network latencies and invalidation times when

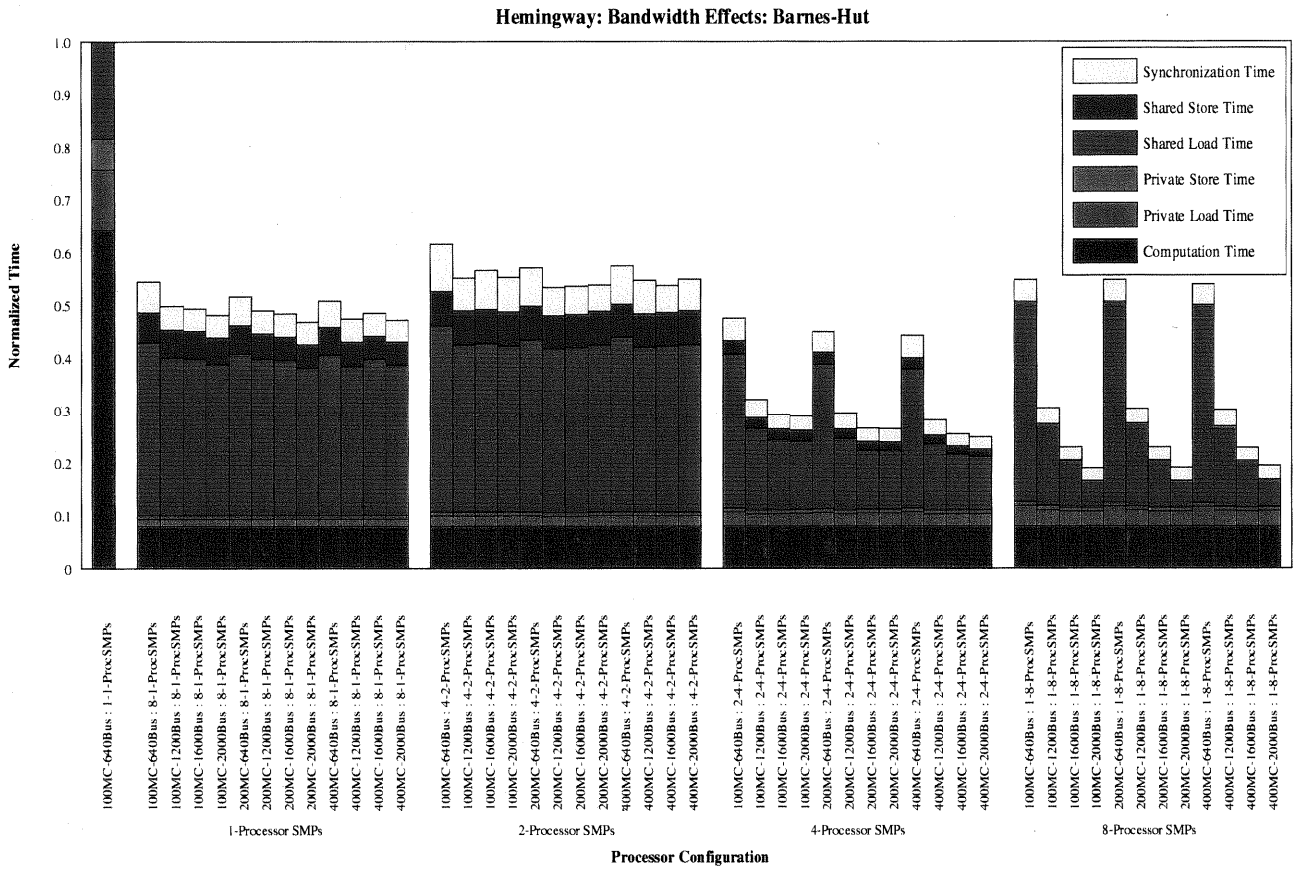


Figure 4: Effects of network bandwidths on performance for Barnes-Hut with Hemingway protocol.

the sharing granularity is small [10]. Multi-writer write-back systems do not suffer from false sharing or excessive network and invalidation latencies, but may have high-latency synchronization operations due to the computation of expensive "diffs". Multi-writer write-through implementations alleviate, additionally, the high-latency synchronization operations [12, 26].

Three major multi-writer, write-through systems have been developed to date, Shrimp [6], Cashmere [15] and Hemingway [2]. The Cashmere and Hemingway protocols focus on *emulation* of write-through protocols with eventual write-through implementation by binary modification, since the existing interconnect used by both projects (Memory Channel) does not support write-through in practice. The Shrimp protocol supports simultaneous update of local and remote memory [5], thus performing a *true* write-through protocol using their hardware. Cashmere and Hemingway differ in that the Hemingway protocol was originally designed for multiprocessors, while the Cashmere protocol was designed for uniprocessors. In addition, the Cashmere design relied on the use of ECC bits to support sharing at the cache-line level. Because of the lack of implementational feasibility of such a design, however, the approach was recently changed by increasing the sharing granularity to a page. Cashmere is currently being implemented on networked workstations, but does not use the SMP hardware coherence mechanism for clustering - instead each processor is treated as a separate node.

The MGS system [25] developed on top of MIT's Alewife machine [1], explored the benefit of *multigrain sharing and locality*. This work differs from ours in two aspects. First, MGS is built out of specialized Alewife processor nodes, each with its own routing chip, communications unit, memory management unit and hardware support for low overhead multithreading, while our system is built out of commodity workstations. Second, MGS uses the underlying Alewife communication network for messages between "clusters". The measurements and results do not take into account network contention. The main problem with building software DSMs out of multiprocessor workstations

is the increase in network contention and traffic with the increase in the number of processors per workstation [9]. Network congestion is the dominant cause for a decrease in performance with multiprocessor workstations. Tolerating and decreasing communication latency is one of Alewife's greatest strengths. A software DSM system built on top of that machine will not perform in the same way as a software DSM system built out of commodity workstations.

Many other software DSM systems have been constructed to date. Some, such as Midway [4] and CRL [13] require programmer annotations to identify synchronization variables with shared variables. We have limited our discussion to modifications of DSM system parameters for effecient execution of existing shared memory programs.

6 Conclusion

With the increase in availability of commodity multiprocessor workstations arises the question of the feasability of the use of such workstations in the design of software DSM systems. This paper addresses this issue by investigating the performance of the Hemingway system on a variety of platforms consisting of different sized-multiprocessor workstations connected together by networks of different bandwidths. The performance of Hemingway protocol was compared to the performance of a similar, established protocol, the Munin protocol.

The three main aspects of performance that were addressed were the scalability of the protocol with processor size, the scalability with SMP-size and the network bandwidth requirements. We found that Hemingway achieved near-linear speedup on 3 of the 6 applications, with just an increase in the total number of processors. The performance increased even further with clustering in all of the applications, except Water. Water did not scale with clustering because computation dominated the total execution time, and clustering, with a non-linear increase in intra-SMP network bandwidth, lead to an increase in the latencies of the private memory operations. Finally, we explored the effects of network bandwidths on system performance. As cluster-size increased, the effect of inter-cluster bandwidth decreased, but the effect of intra-cluster bandwidth increased. However, doubling the intra-cluster bandwidth (from 640MB/s to 1.2GB/s) in increasing cluster size by a factor of even four or eight improved performance by between 30 to 50 percent.

We conclude, then, that small-scale shared memory multiprocessors can be used very effectively to build software distributed shared memory systems. Commodity multiprocessor workstations, with commodity communication networks have enough network bandwidths (for example 1.2GB/s backplance in Digital's 8400s) to satisfy scaling requirements. Previous work has indicated that single-writer protocols are not effective in the design of DSM systems based on SMPs. We further conclude that multiwriter write-through memory consistency policies are very effective and appropriate for the design of software DSMs based on small-scale SMPs.

The positive results from the simulations have motivated the implementation of Hemingway. We have at present, a write-back protocol which has been verified on simple test cases. We are now modifying this protocol to implement write-through. Once this is complete we will make use of the underlying Shasta framework of the implementation to explore the benefits of fine-grain sharing in SMP-based software DSM systems.

Acknowledgments

We would like to thank Dan Scales, Kourosh Gharachorloo and Marco Annaratone for making the Shasta framework available to us for the development of Hemingway. This work was funded in part by NSF grant No. ASC-9217394, ARPA contract ARMY DABT63-94-C-0029 and an equipment grant from Digital Equipment Corporation.

References

[1] A. Agarwal, *et al.* The MIT Alewife Machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, 1995.

[2] A. Aggarwal, D. Grunwald, T. Hein, and E. Nemeth. Hemingway, a distributed shared memory system. CU-CS 813-96, University of Colorado, Boulder, 1996.

[3] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 168–176, 1990.

[4] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON '93)*, pages 528–537, February 1993.

[5] A. Bilas, L. Iftode, and J. P. Singh. Shared virtual memory across SMP nodes using automatic update: Protocols and performance. Sixth Workshop on Shared-Memory Multiprocessors, October 1996.

[6] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felton, and J. Sandberg. Virtual memory mapped network interface for the shrimp multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–154, April 1994.

[7] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepol. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[8] Digital Equipment Corperation, Maynard, Mass. *DECchip 21064 Microprocessor: Hardware Reference Manual*, October 1992.

[9] A. Erlichson, B. A. Nayfeh, J. P. Singh, and K. Olukotun. The benefits of clustering in shared address space multiprocessors: An applications-driven investigation. Technical Report CSL-TR-94-632, Stanford University, 1994.

[10] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the performance of clustered distributed shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, 1996.

[11] R.B. Gillett. Memory Channel Network for pci. *IEEE Micro*, 16(1):12–18, February 1996.

[12] L. Iftode, J. P. Singh, and K. Li. Understanding application performance on shared virtual memory systems. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 122–133, 1996.

[13] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[14] P. Keleher. The relative importance of concurrent writers and weak consistency models. CS-TR 3543, University of Maryland, 1995.

[15] L. Kontothanassis and M. L. Scott. Software cache coherence for large scale multiprocessors. In *Proceedings of 1st Conference on High Performance Computer Architecture*, January 1995.

[16] N. P. Kronenberg, H. Levy, and W. D. Strecker. VAXclusters: a closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[18] A. Paithankar. AINT: A tool for simulation of shared-memory multiprocessors. Master's thesis, Department of Computer Science, University of Colorado at Boulder, December 1995.

[19] D. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on smp clusters. Technical Report 97/3, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[20] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, 1996.

[21] M. L. Scott, W. Li, S. Dwarkadas, L. Kontothanassis, G. Hunt, M. Michael, R. Stets, N. Hardavellas, W. Meira, A. Poulos, M. Cierniak, S. Parthasarathy, and M. Zaki. Implementation of Cashmere. Sixth Workshop on Shared-Memory Multiprocessors, October 1996.

[22] J. E. Veenstra and R. J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 201–207, 1994.

[23] J. Wilkes. Hamlyn—an interface for sender-based communications. HPL-OSR 92-13, Hewlett-Packard Research Labs, November 1992. Available from ftp.hpl.hp.com.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[25] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A multigrain shared memory system. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 44–55, 1996.

[26] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.